Graph Neural Networks and Node Classification

Prof. O-Joun Lee

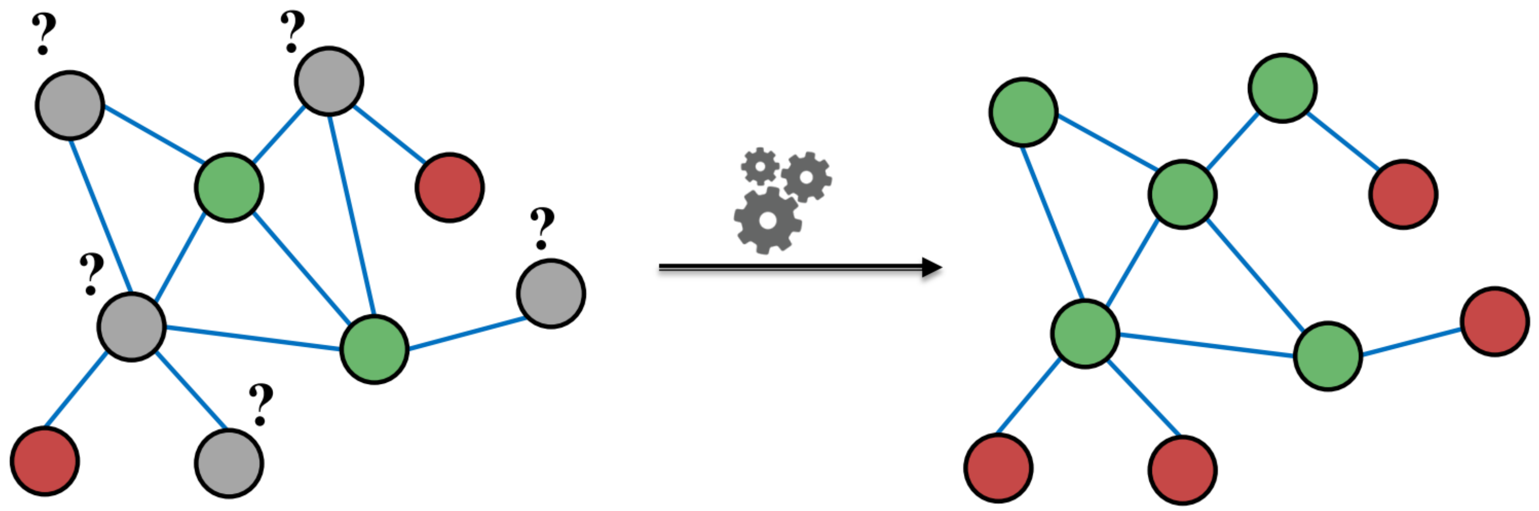
Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실 NETWORK SCIENCE LAB 가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA

Contents

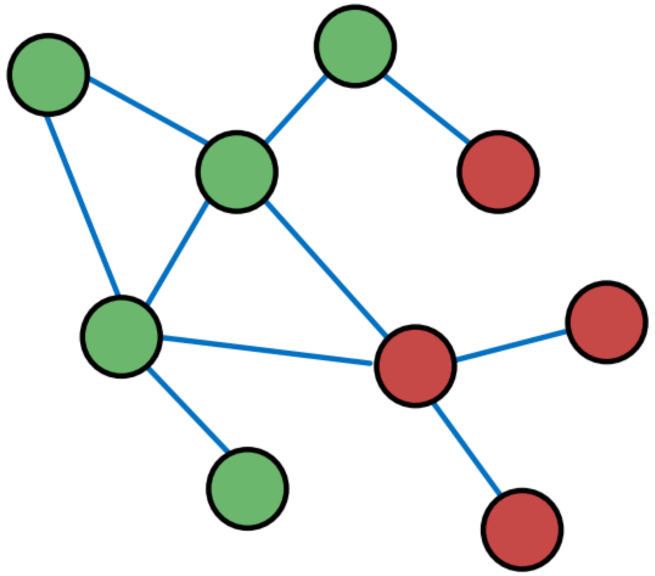
Main Question

- Given a network with labels on some nodes, how do we assign labels to all other nodes in the network?
- Node embedding (shallow encoding) is a method to solve this
- Message passing is an alternative framework



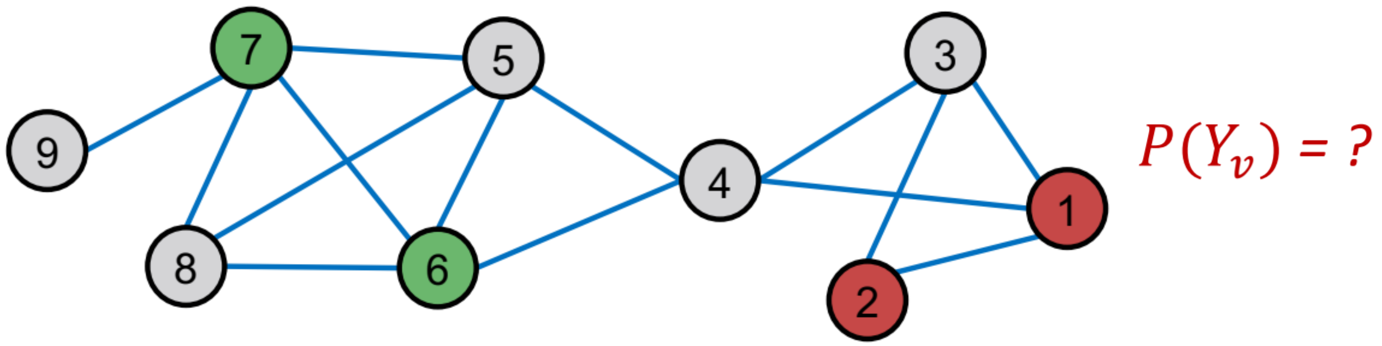
Message Passing Intuition

- **Intuition**: Correlations (dependency) exist in network
  - Similar nodes are connected
  - Homophily: individual character → social connections
  - Influence: social connections → individual character
- **Key concept**: collective classification, assign labels to all nodes in a network together
- We will look at 2 **techniques**:
  - Relational classification
  - Iterative classification



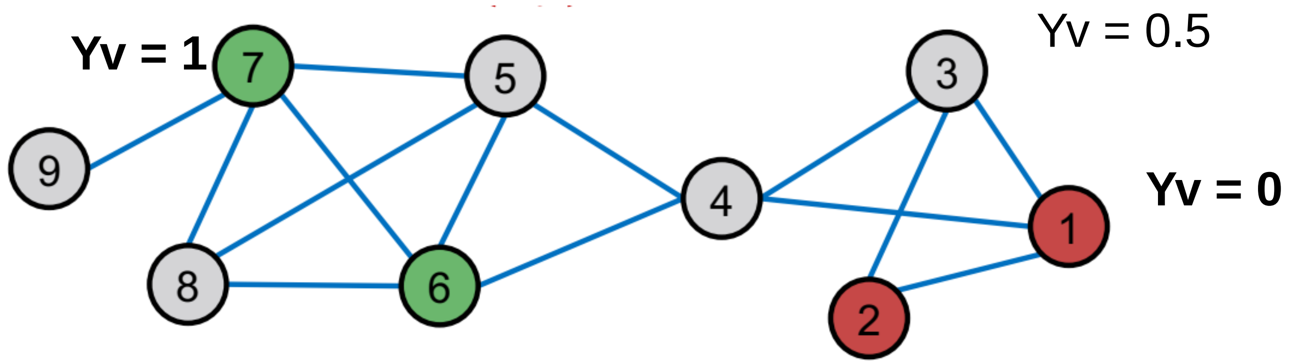
Classification with Network Data

- How do we leverage node correlations in networks to help prediction?
- Motivation 1: Similar nodes are typically close together or directly connected in the network
- Motivation 2: Classification label of node v in network may depend on:
  - features of v
  - labels of the nodes in v ‘s neighbourhood
  - features of the nodes in v ‘s neighbourhood
- Many applications are under the same setting



Relational Classification

- Idea: Propagate node labels across the network
  - Class probability Yv of node v is a weighted average of class probabilities of its neighbors.
- Labeled nodes: initialize Yv with ground-truth label Yv*
- Unlabeled nodes: initialize Yv = 0.5
- Update all nodes in a random order until convergence or until maximum number of iterations.

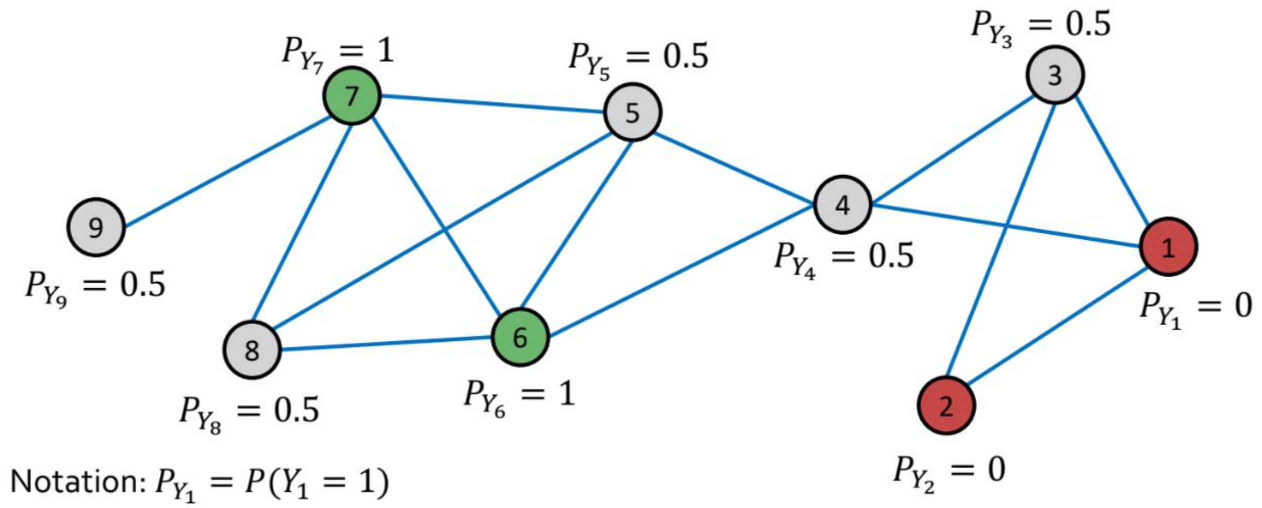


Update for Relational Classifier

- Update for each node v and label c (0 or 1)

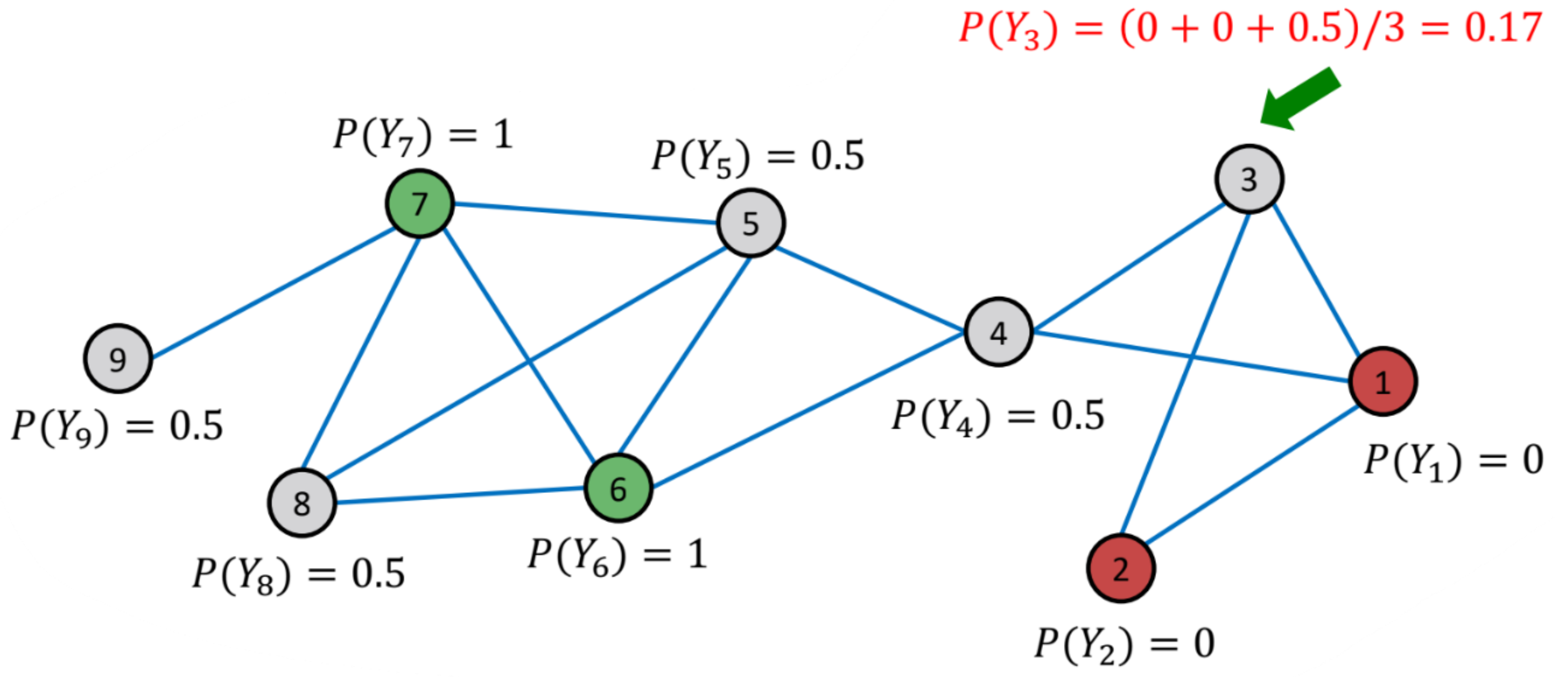
$$P(Y_v = c) = \frac{1}{\sum_{(v,u)\in E} A_{v,u}} \sum_{(v,u)\in E} A_{v,u}\, P(Y_u = c)$$

- If edges have weight/strength, Avu can represent the edge weight
- For example:



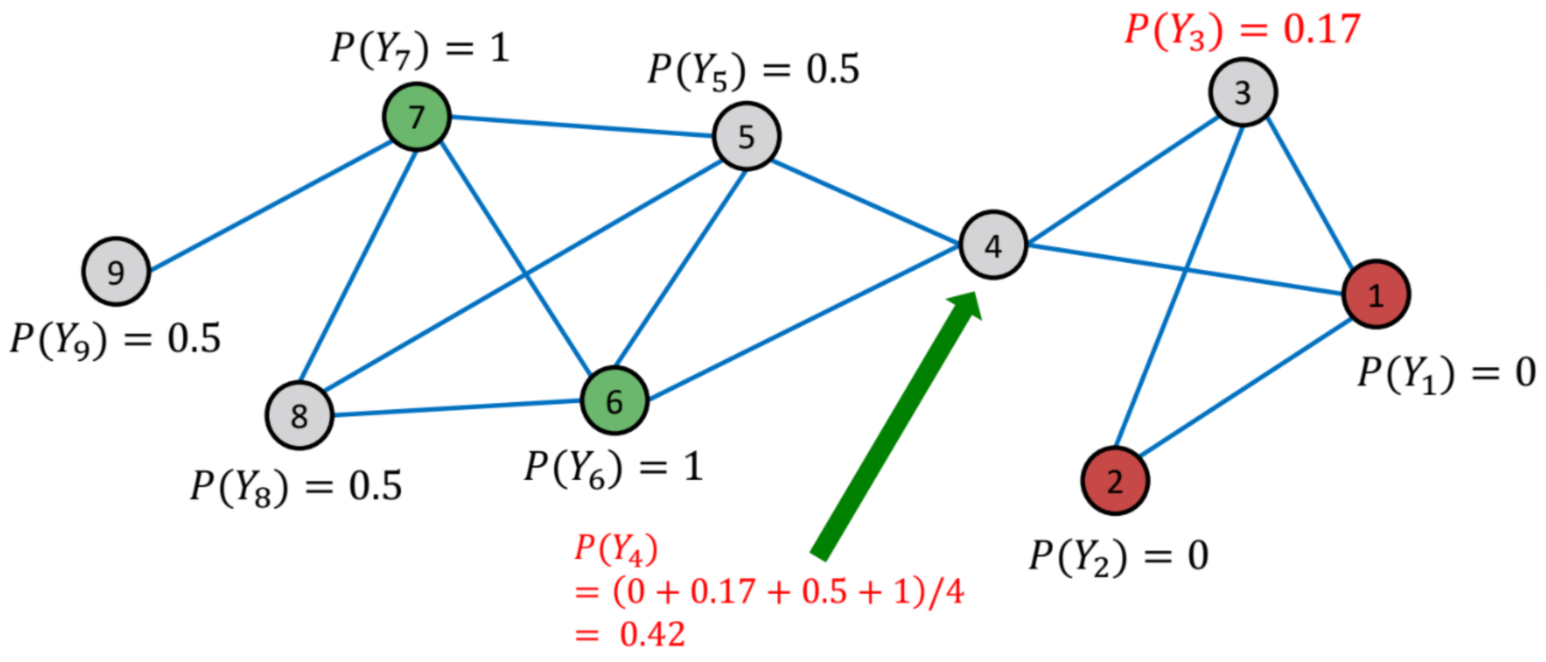
Example: 1st Iteration, Update Node 3

- Update for the 1st iteration:
  - For node 3, N3 = {1, 2, 4}



Example: 1st Iteration, Update Node 4

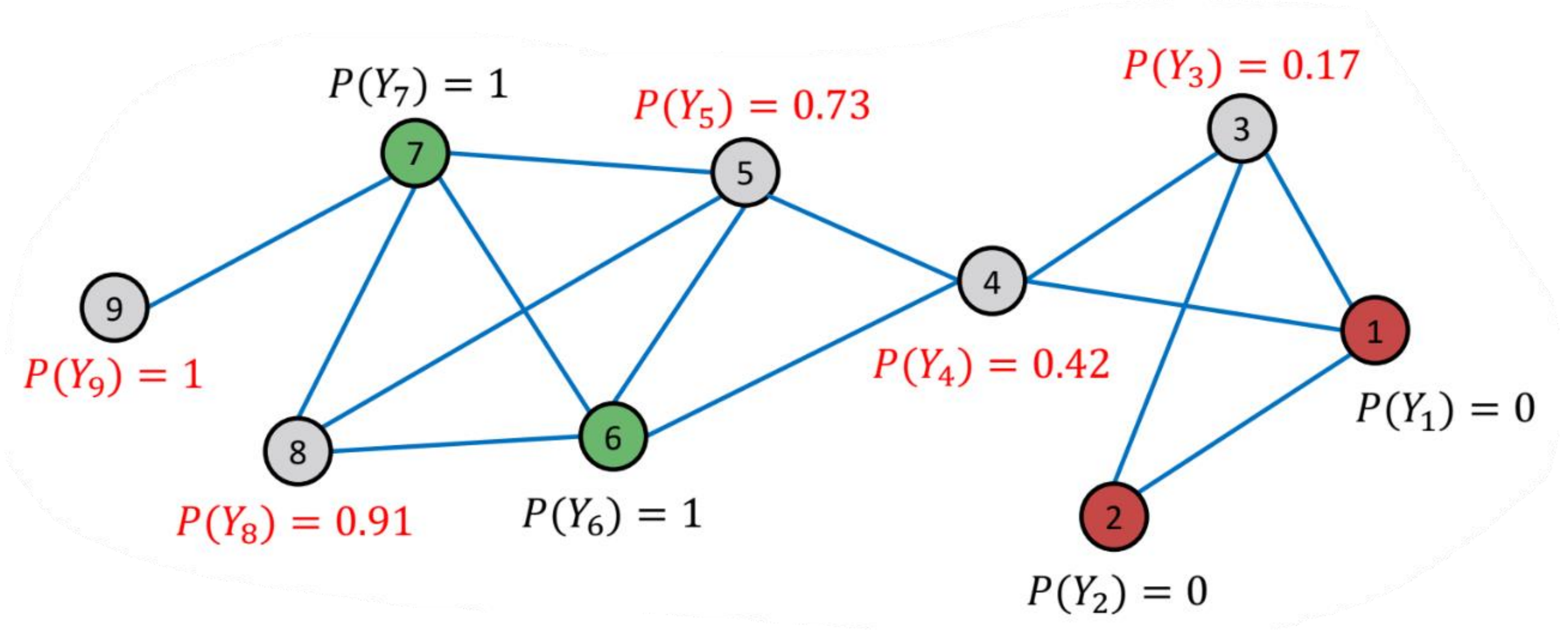
- Update for the 1st iteration:
  - For node 3, N3 = {1, 2, 4}



After Node 3 is updated

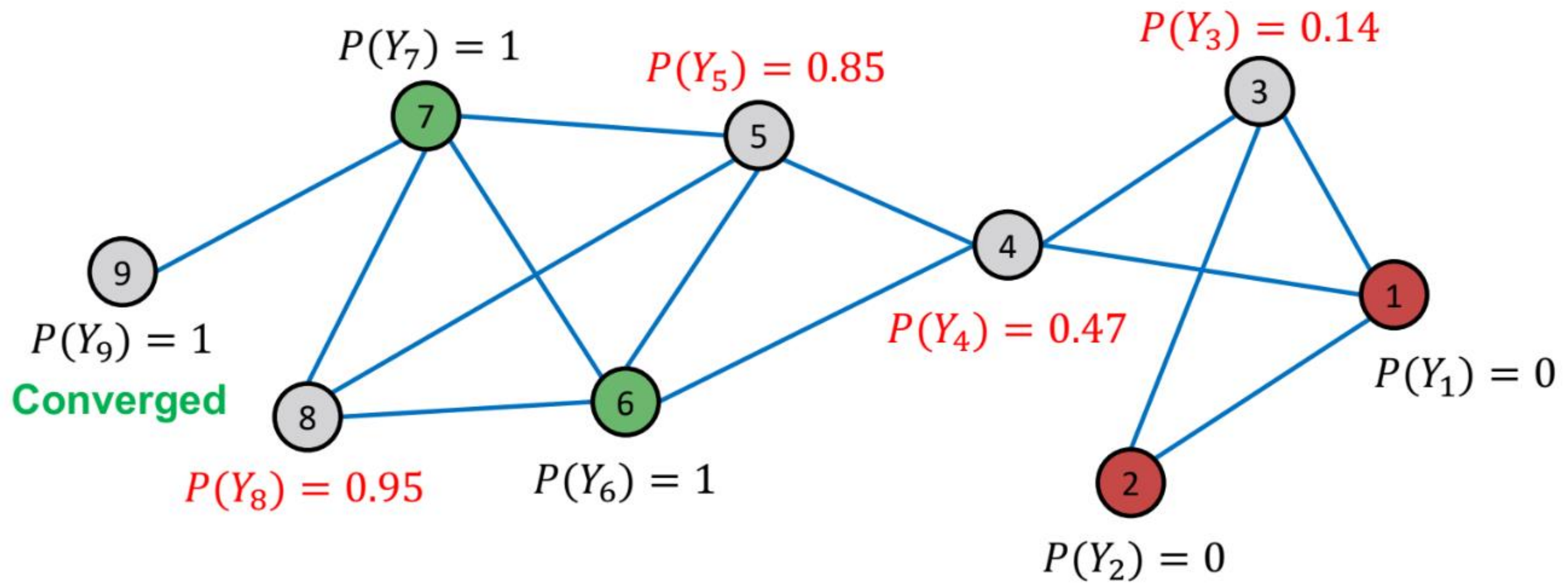
Example: After 1st Iteration

- Then after update for node 5, 8, 9
  - 1 iteration



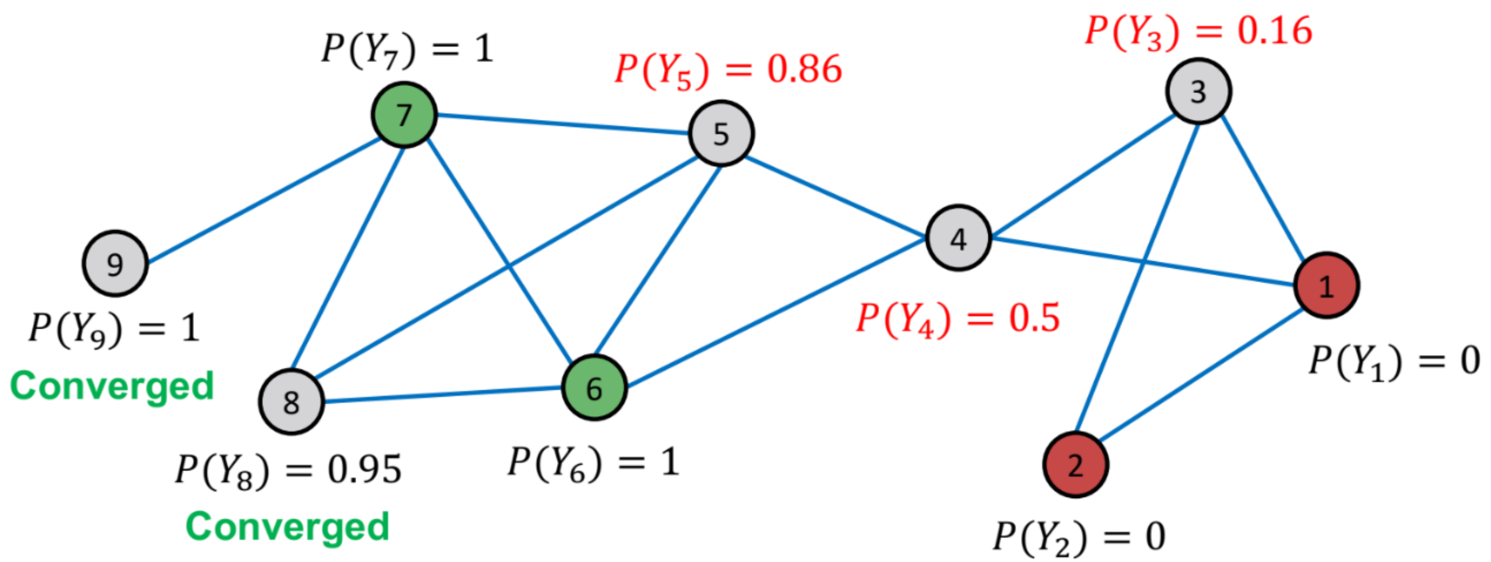
Example: After 2nd Iteration

- Update nodes with another random order
  - After 2nd iteration:



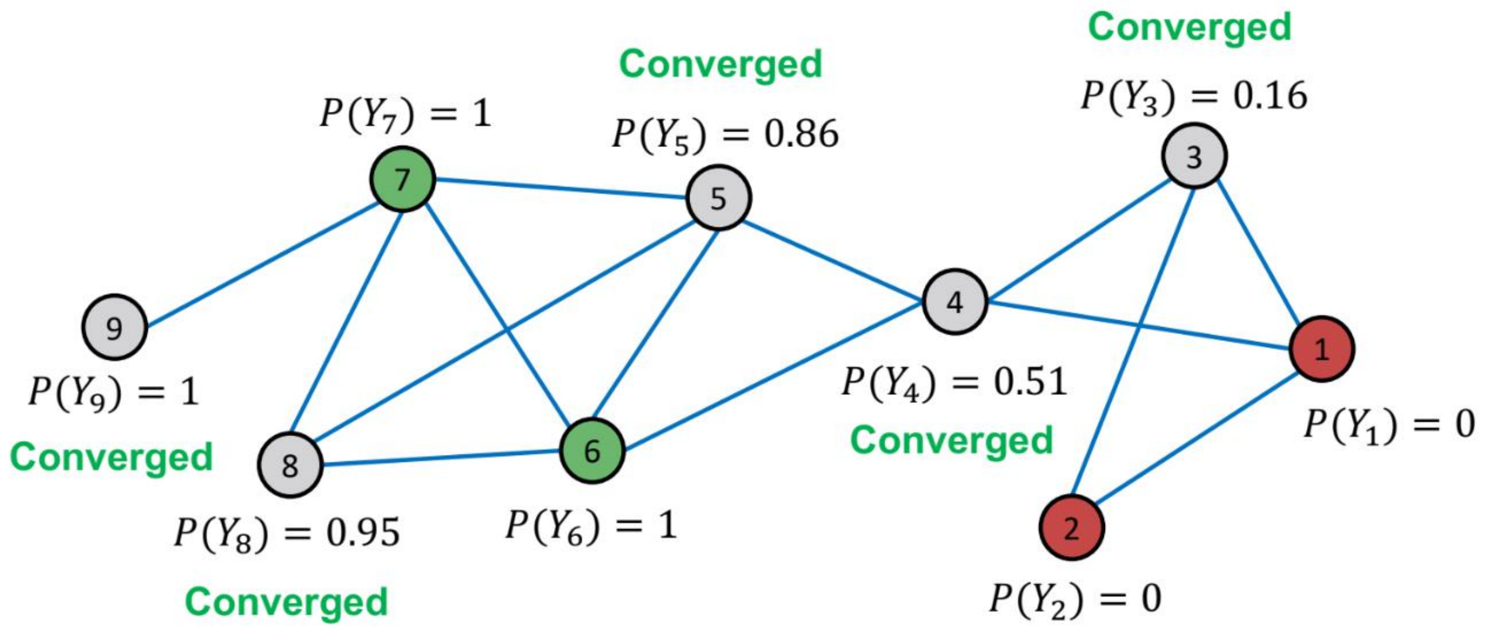
Example: After 3rd Iteration

- Update nodes with another random order
  - After 3rd iteration:



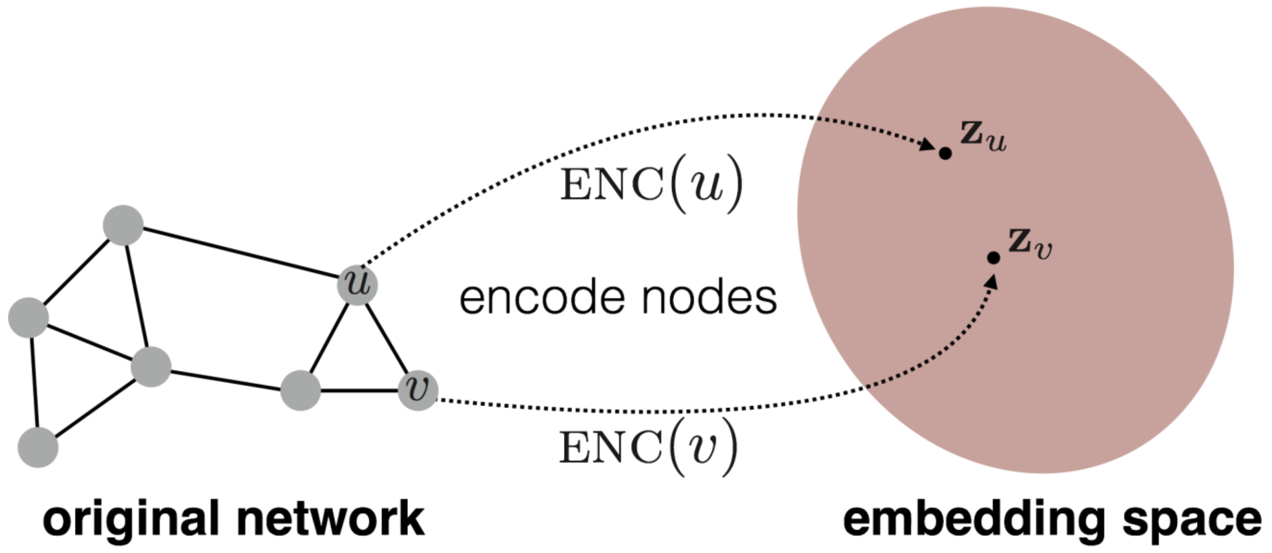
Example: Convergence

- After 4th iteration, it converged, we therefore predict:
  - Nodes 4,5,8,9 belong to Class 1 $(P_{Y_v} > 0.5)$
  - Node 3 belong to Class 0 $(P_{Y_v} < 0.5)$
- **Challenges:**
  (1) Convergence not guaranteed
  (2) Cannot use node feature information



ReCap: Embedding Nodes

- Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network.

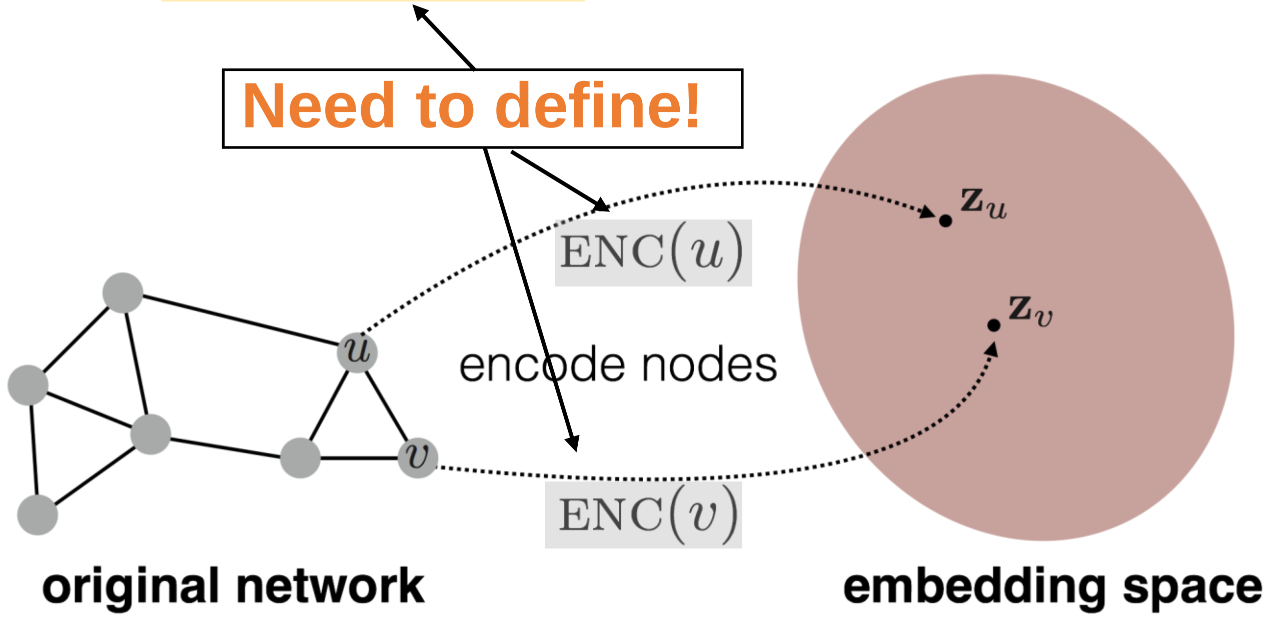


ReCap: Embedding Nodes

- Goal:

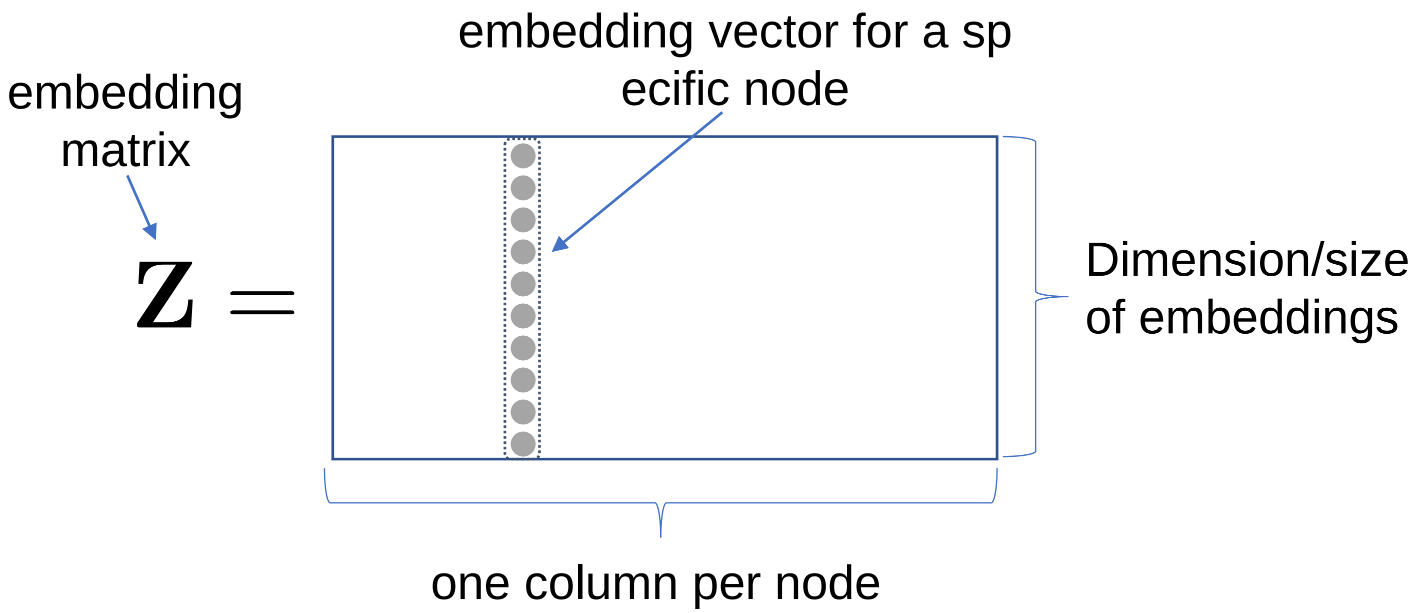
$$\text{similarity}(u, v) \approx \mathbf{z}_v^\top \mathbf{z}_u$$

Need to define!



From “Shallow” to “Deep”

- So far we have focused on “shallow” encoders, i.e. embedding lookups:


embedding matrix
$\mathbf{Z} =$
embedding vector for a specific node
Dimension/size of embeddings
one column per node

From “Shallow” to “Deep”

- Limitations of shallow encoding:
  - O(|V|) parameters are needed: there no parameter sharing and every node has its own unique embedding vector.
  - Inherently “transductive”: It is impossible to generate embeddings for nodes that were not seen during training.
  - Do not incorporate node features: Many graphs have features that we can and should leverage.

From “Shallow” to “Deep”

- We will now discuss “deeper” methods based on graph neural networks.

$$\mathrm{ENC}(v) =$$ complex function that depend
s on graph structure.

- In general, all of these more complex encoders can be combined with the similarity functions from the previous section.

The Basics: Graph Neural Networks

- **Idea**: Propagate node features by exchanging info between adjacent nodes.
- Message:
  - Prepare message that will be passed to another nodes
- Aggregate and Update
  - Aggregate the coming messages
  - Update node with the aggregated message

Based on material from:

Hamilton et al. 2017. Representation Learning on Graphs: Methods and Applications. IEEE Data Engineering Bulletin on Graph Systems.

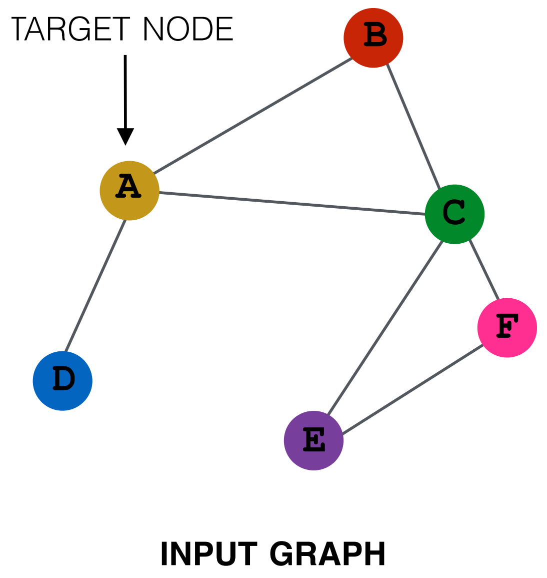
Scarselli et al. 2005. The Graph Neural Network Model. IEEE Transactions on Neural Networks.

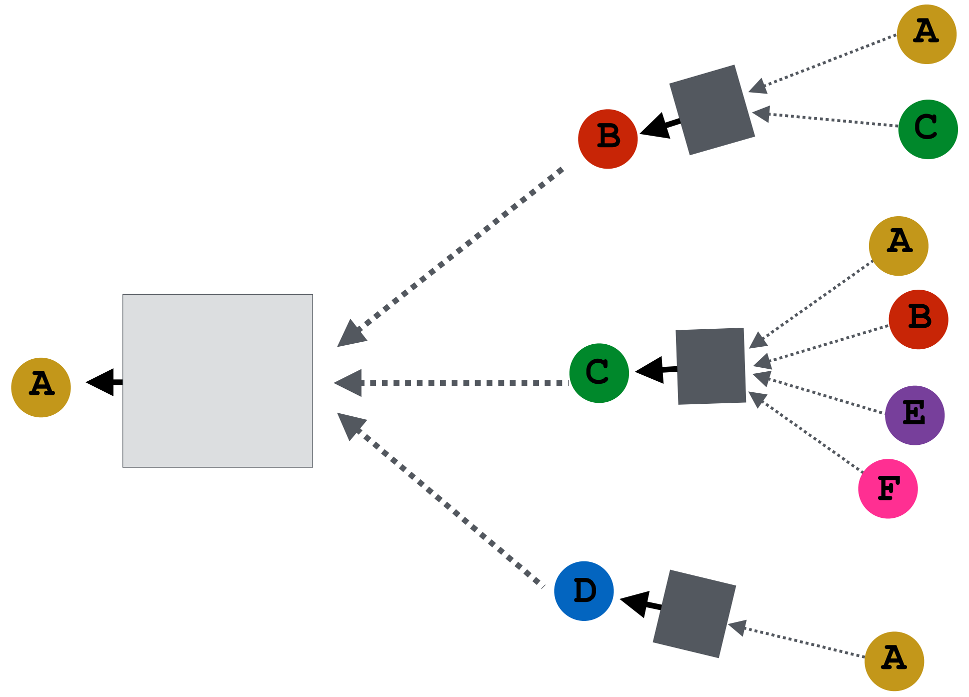
Graph Neural Networks: Setup

- Assume we have a graph G:
  - V is the vertex set.
  - A is the adjacency matrix (assume binary).
  - $X \in R^{\mathrm{N} \times d}$ is a matrix of node features.
    - Categorical attributes, text, image data
    - E.g., profile information in a social network.
    - Node degrees, clustering coefficients, etc.
    - Indicator vectors (i.e., one-hot encoding of each node)

Neighborhood Aggregation

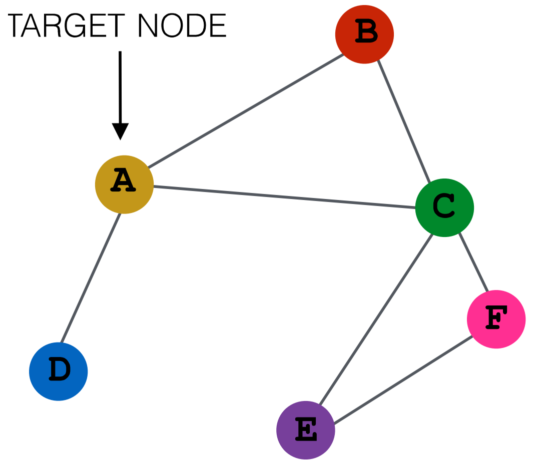
- Key idea: Generate node embeddings based on local neighborhoods.



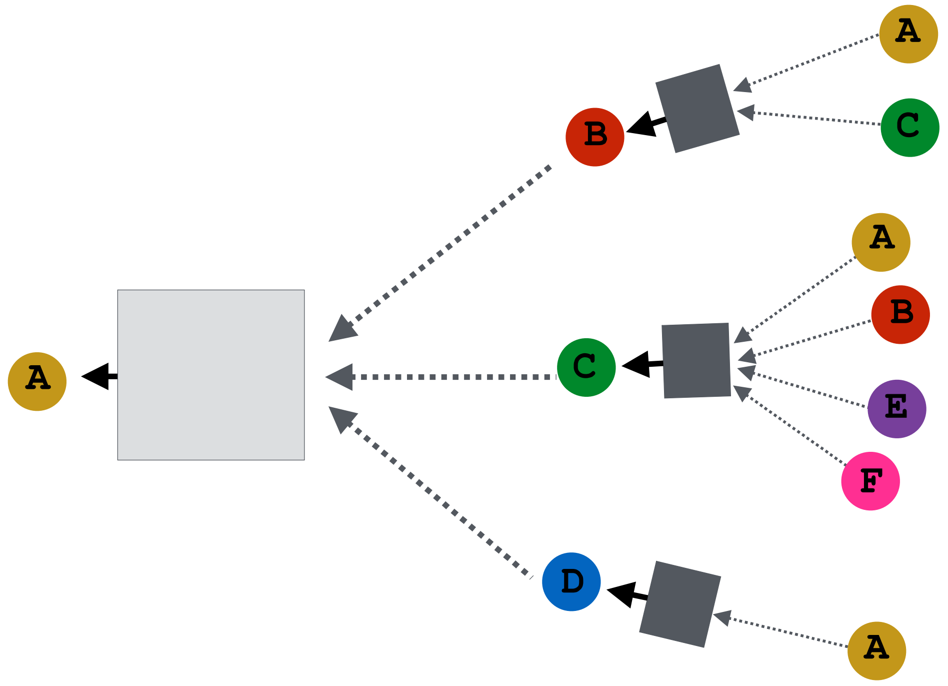


Neighborhood Aggregation

- Intuition: Nodes aggregate information from their neighbors using neural networks

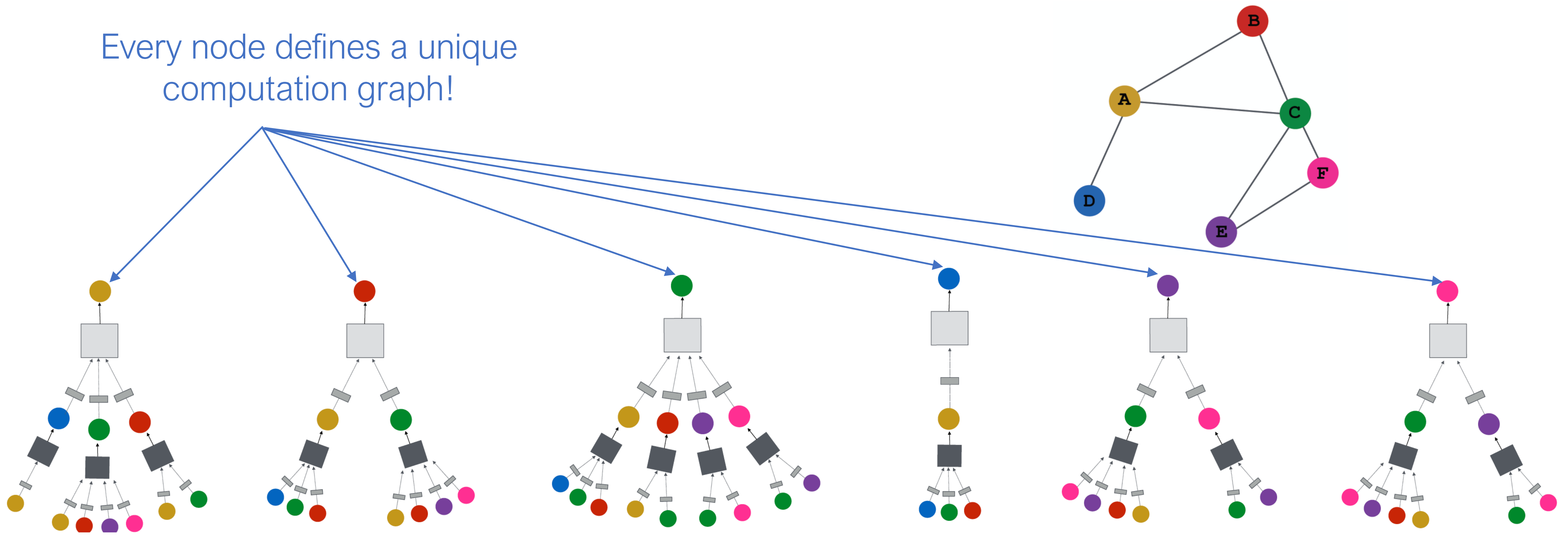


INPUT GRAPH



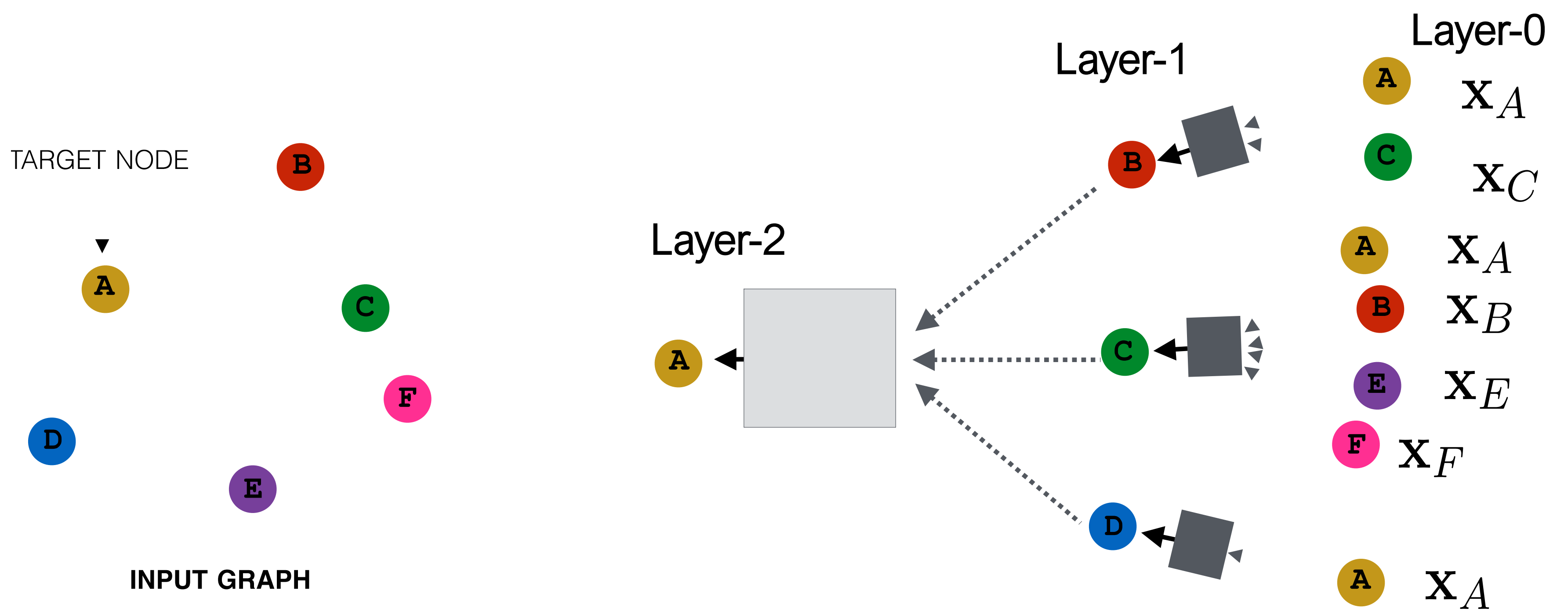
Neighborhood Aggregation

➢ Intuition: Network neighborhood defines a computation graph



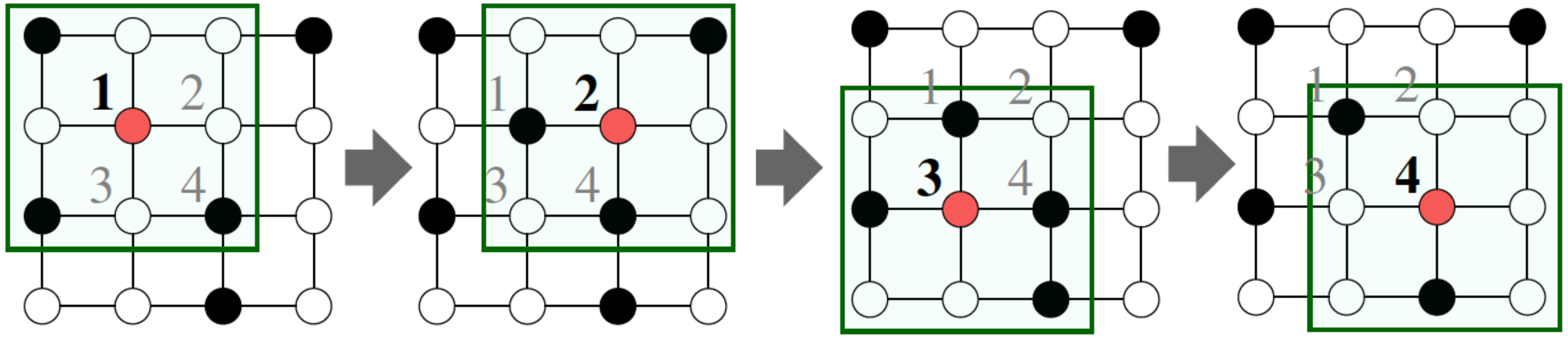
Neighborhood Aggregation

- Nodes have embeddings at each layer.
- Model can be arbitrary depth.
- “layer-0” embedding of node u is its input feature, i.e. x_u



Neighborhood “Convolutions”

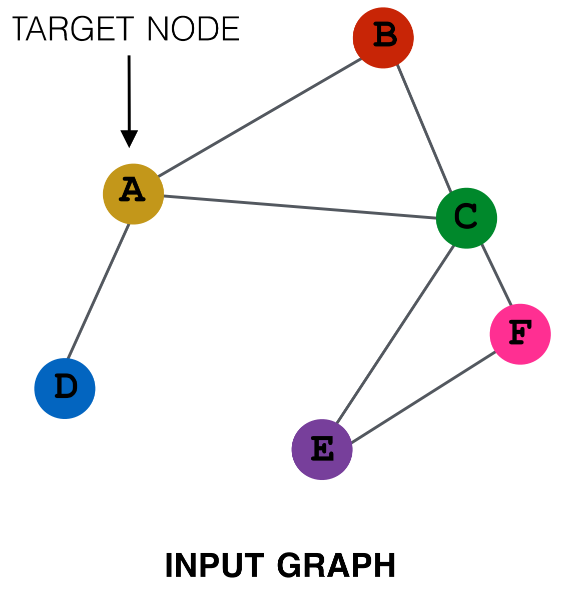
- Neighborhood aggregation can be viewed as a center-surround filter.

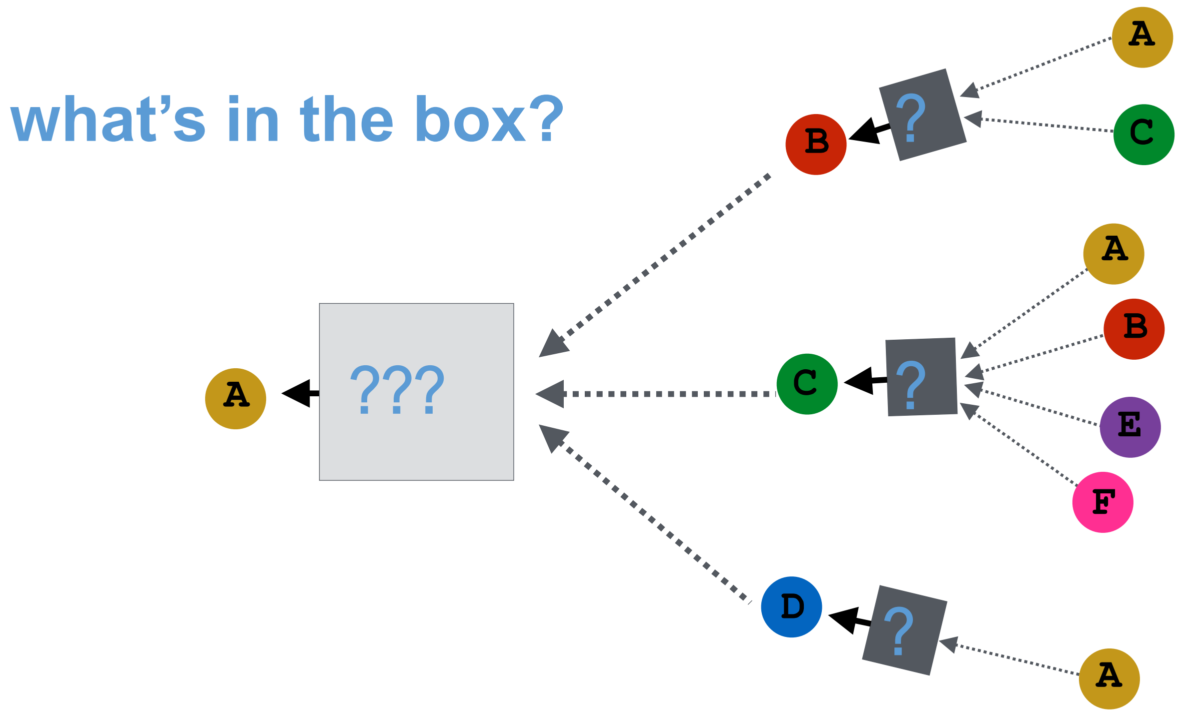


- Mathematically related to spectral graph convolutions

Neighborhood Aggregation

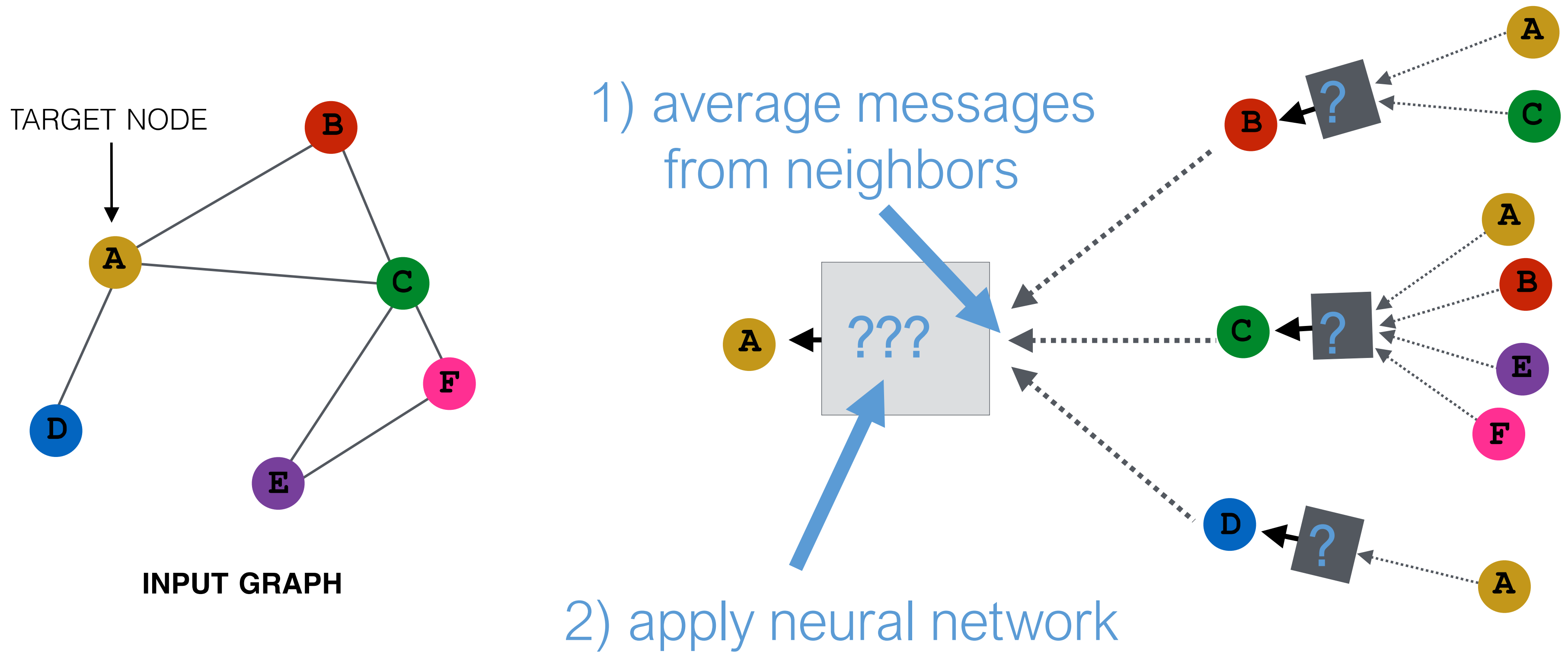
➢ Key distinctions are in how different approaches aggregate information across the layers.





Neighborhood Aggregation

- Basic approach: Average neighbor information and apply a neural network.



Neighborhood Aggregation: The Math

➢ Basic approach: Average neighbor messages and apply a neural network.

Initial “layer 0” embeddings are equal to node features

$$\boxed{\mathbf{h}_v^0 = \mathbf{x}_v}$$

previous layer embedding of v

$$\boxed{\mathbf{h}_v^k} = \boxed{\sigma}\left(\mathbf{W}_k \boxed{\sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}} + \mathbf{B}_k \boxed{\mathbf{h}_v^{k-1}}\right), \quad \forall k > 0$$

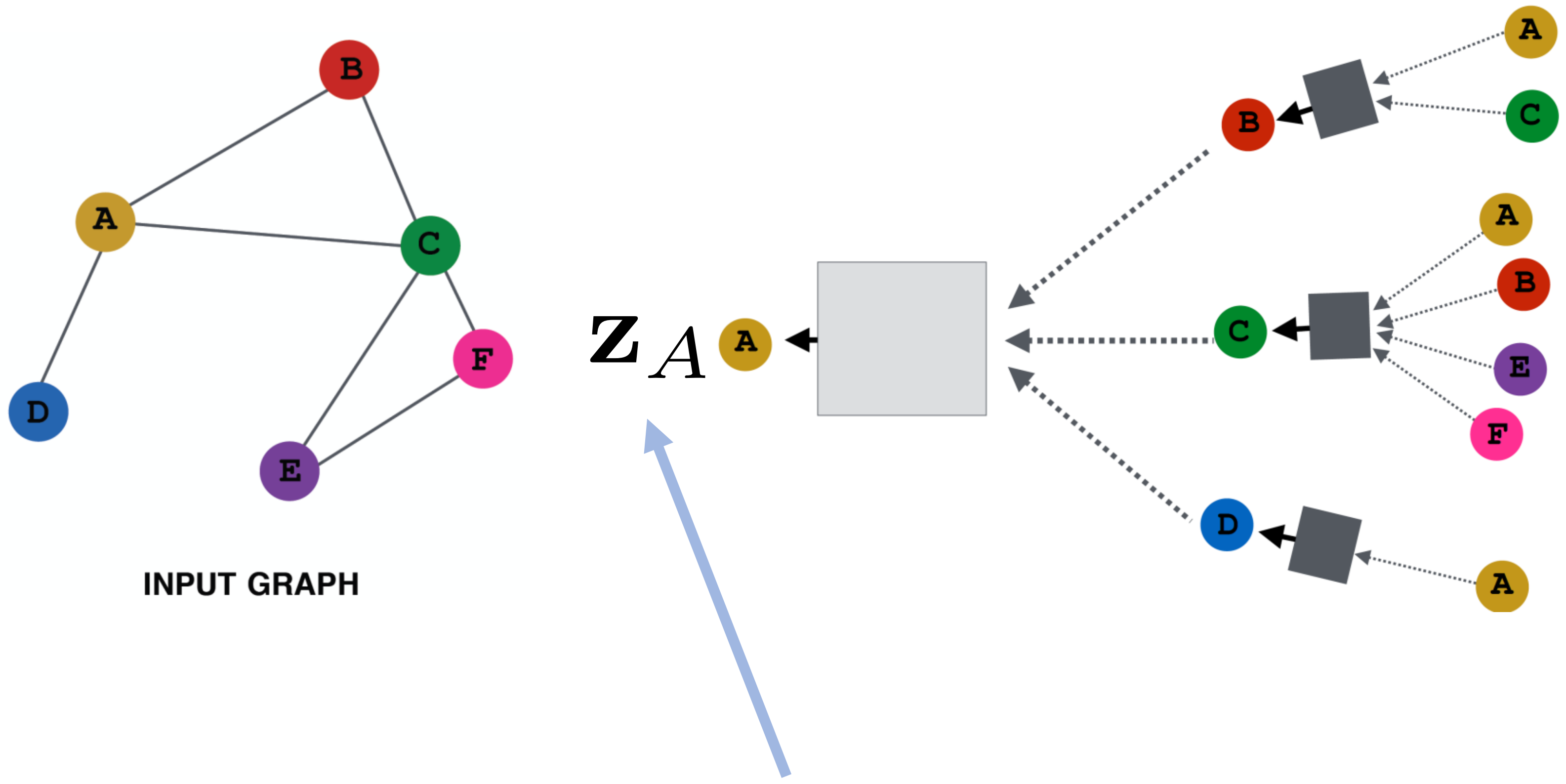
kth layer embedding of v

non-linearity (e.g., ReLU or tanh)

average of neighbor’s previous layer embeddings

Training the Model

➢ How do we train the model to generate “high-quality” embeddings?



Need to define a loss function on the embeddings, L(z_u)!

Training the Model

trainable matrices (i.e., what we learn)

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

$$\mathbf{h}_v^k = \sigma\left(\boxed{\mathbf{W}_k} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \boxed{\mathbf{B}_k} \mathbf{h}_v^{k-1}\right), \ \forall k \in \{1, \ldots, K\}$$

$$\mathbf{z}_v \boxed{= \mathbf{h}_v^K}$$

- After K-layers of neighborhood aggregation, we get output embeddings for each node.
- We can feed these embeddings into any loss function and run stochastic gradient descent to train the aggregation parameters.

Training the Model

- Train in an unsupervised manner using only the graph structure.
- Unsupervised loss function can be anything from the last Lectures, e.g., based on:
  - Random walks (node2vec, DeepWalk)
  - Graph factorization
  - i.e., train the model so that “similar” nodes have similar embeddings.

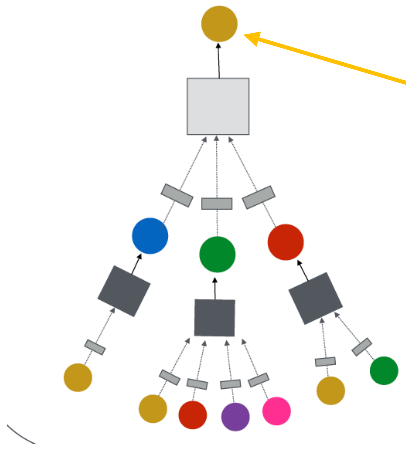
Overview of Model

- Alternative: Directly train the model for a supervised task (e.g., node classification):

classification weights

Human or b
ot?

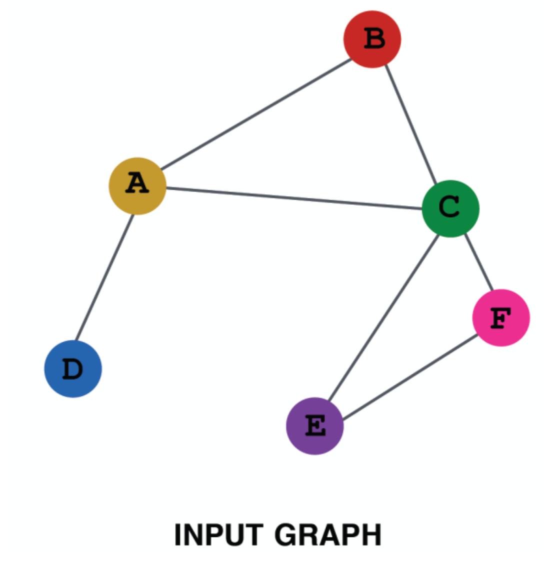
$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(\mathbf{z}_v^\top \boldsymbol{\theta})) + (1 - y_v) \log(1 - \sigma(\mathbf{z}_v^\top \boldsymbol{\theta}))$$



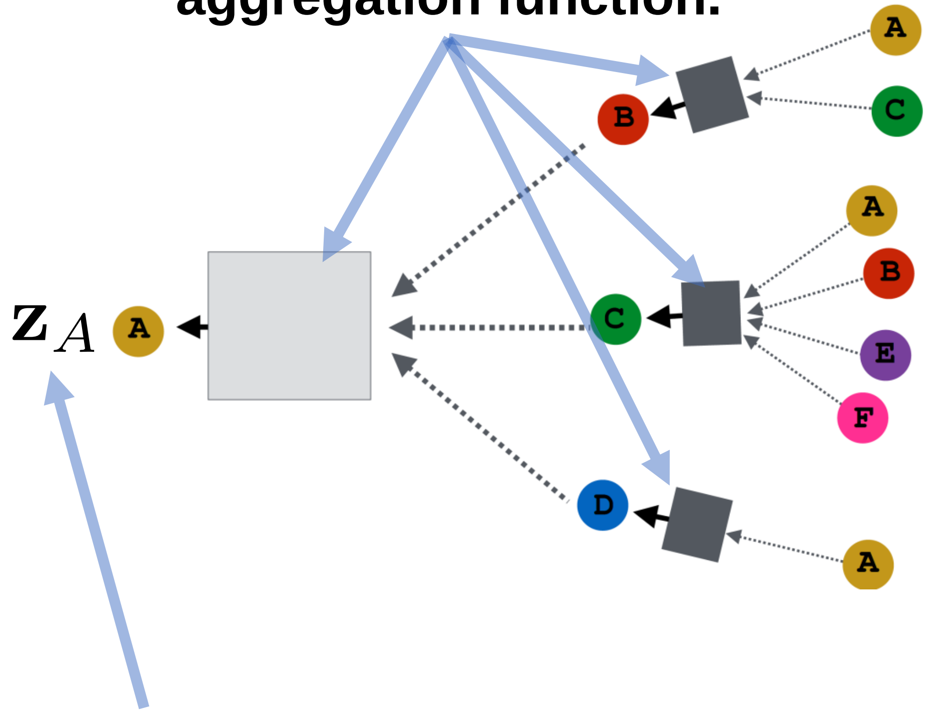
output node embeddin
g

node class label

Overview of Model Design

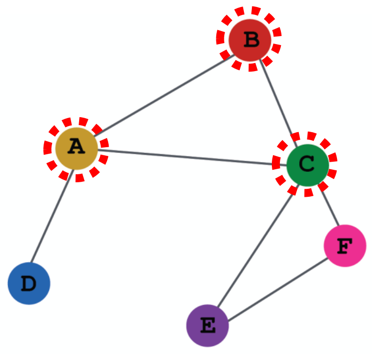


1) Define a neighborhood aggregation function.



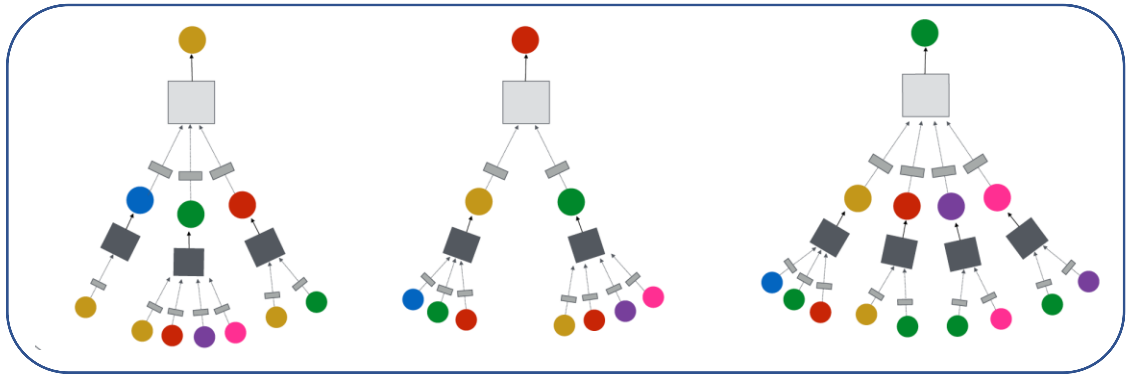
2) Define a loss function on t he embeddings, $L(z_u)$

Overview of Model Design

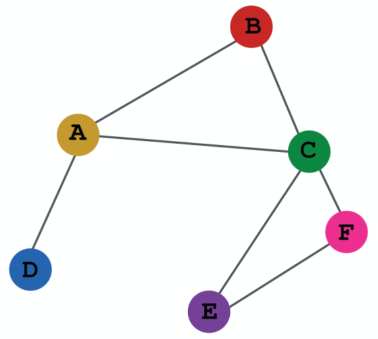


INPUT GRAPH

3) Train on a set of nodes, i.e., a b atch of compute graphs



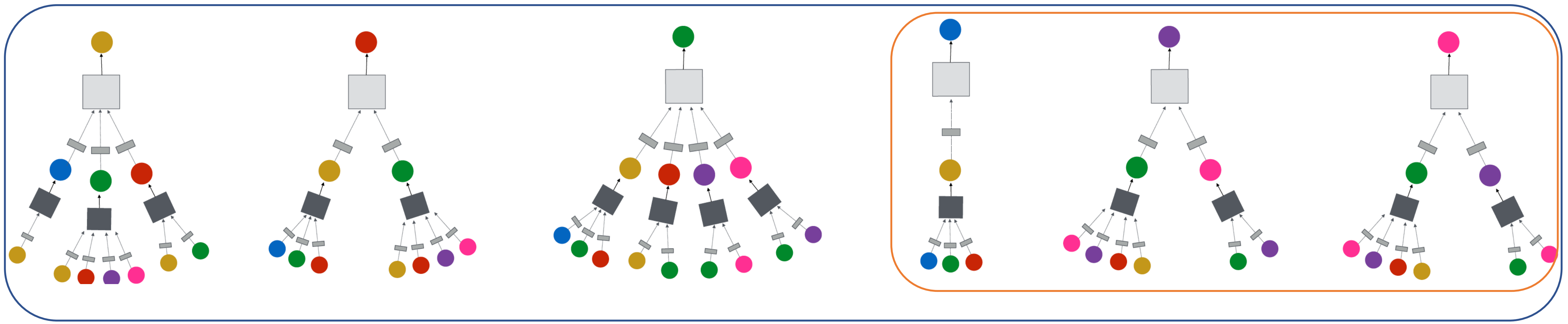
Overview of Model Design



INPUT GRAPH

4) Generate embeddings for nodes as needed

Even for nodes we never trained on!!!!



Graph Neural Network Libraries: Introduction

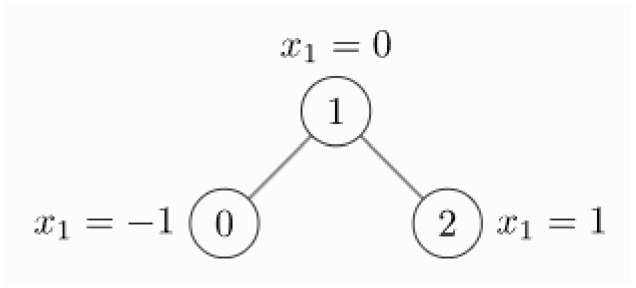
- PyTorch Geometric
  - PyG is a library built upon PyTorch to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data.
  - PyG consists of various methods for deep learning on graphs from a variety of published papers.
- DGL (Deep Graph Library)
  - DGL is a Python package built for easy implementation of graph neural network model family, on top of existing DL frameworks
  - It has a Diverse Ecosystem, including bioinformatics and cheminformatics, and many others.

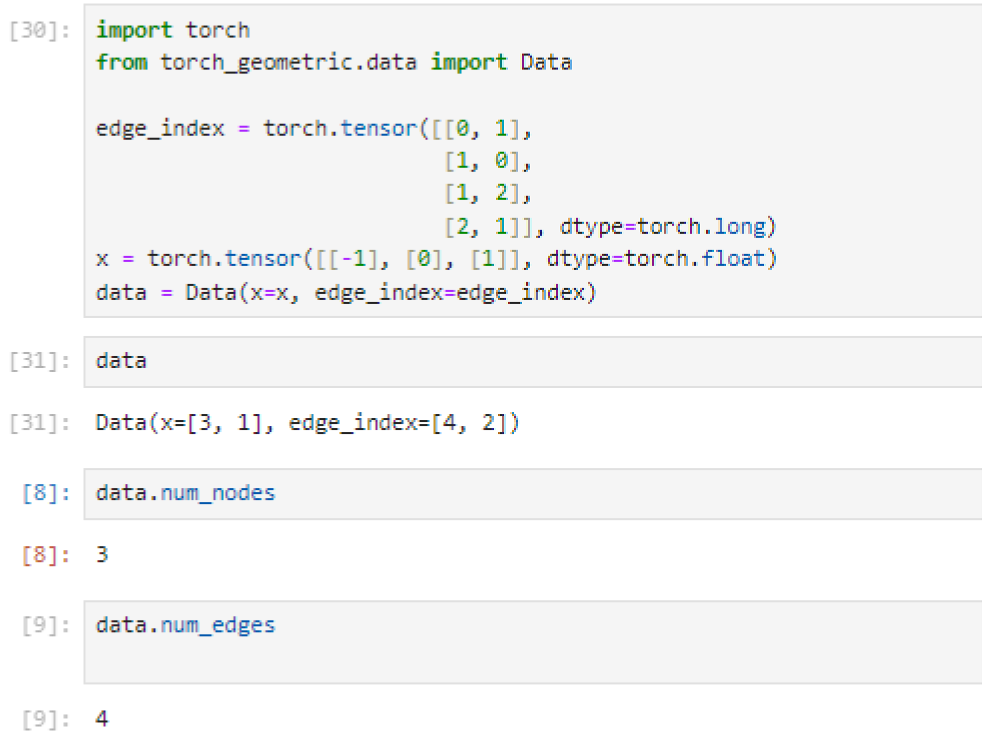
- Note that:
  - Both libraries already contain a lot of useful datasets.
  - Both libraries only represent a graph under edge index

PyTorch Geometric: Creating graph datasets

- Construct a simplified Graph
  - Note that PyG only represent Graph with **edge index**

  E.g., a simple example of an unweighted and undirected graph with three nodes and four edges. Each node contains exactly one feature

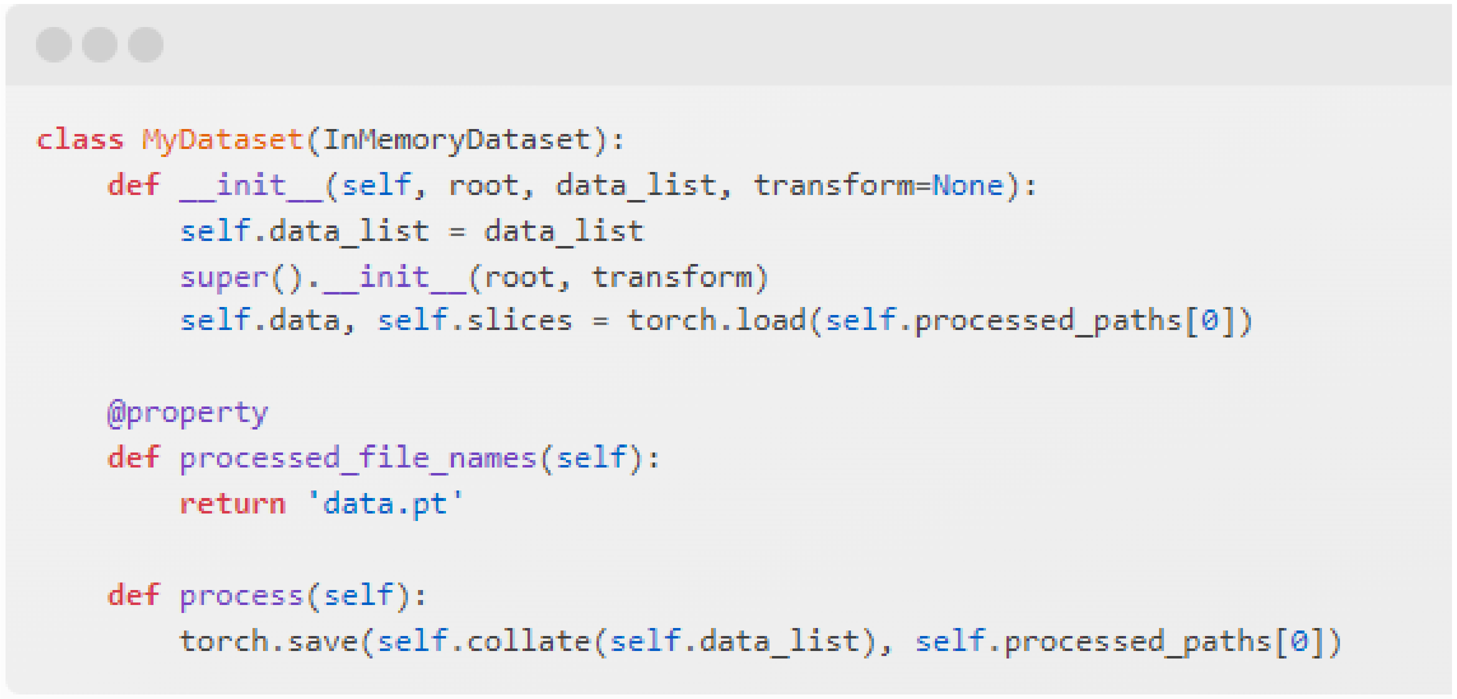




```
[30]: import torch
      from torch_geometric.data import Data

      edge_index = torch.tensor([[0, 1],
                                 [1, 0],
                                 [1, 2],
                                 [2, 1]], dtype=torch.long)
      x = torch.tensor([[-1], [0], [1]], dtype=torch.float)
      data = Data(x=x, edge_index=edge_index)

[31]: data

[31]: Data(x=[3, 1], edge_index=[4, 2])

[8]: data.num_nodes

[8]: 3

[9]: data.num_edges

[9]: 4
```

PyTorch Geometric: Creating our own Graph Datasets

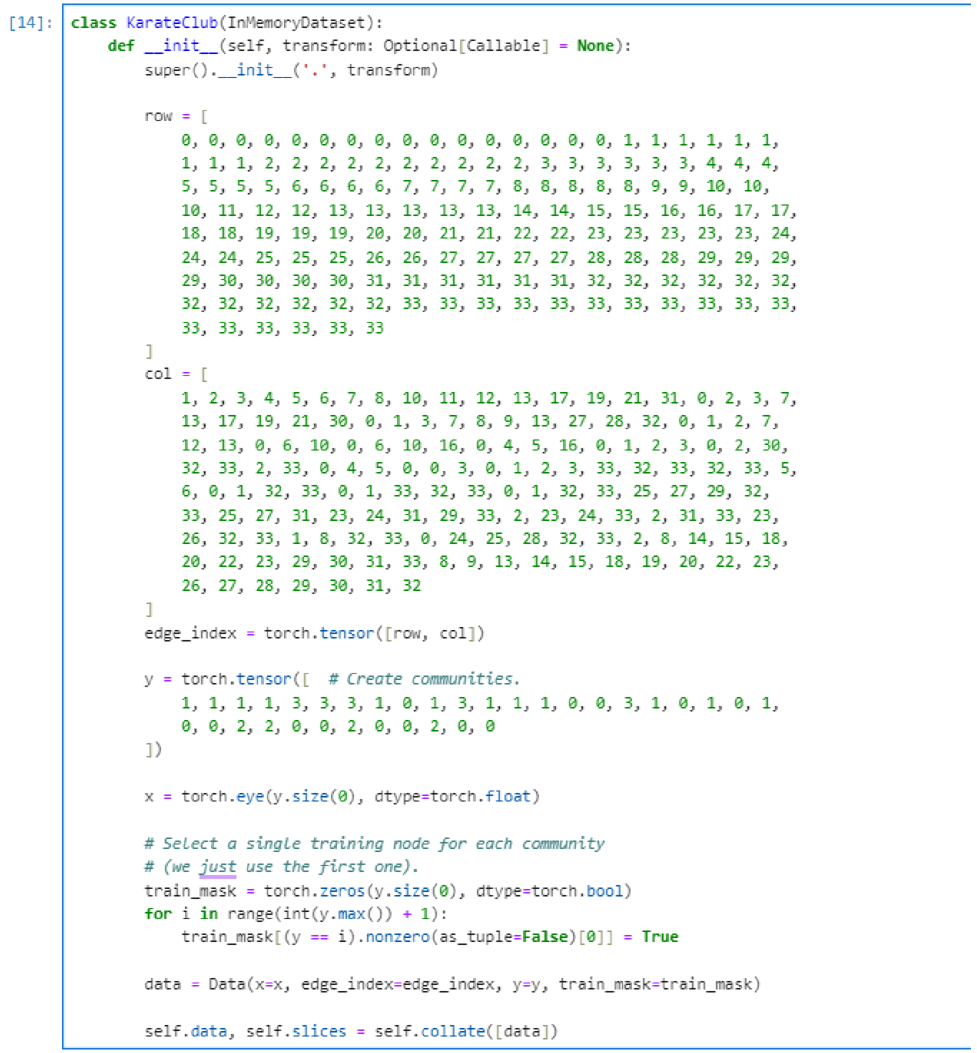
- Although PyG already contains a lot of useful datasets, we can create our own dataset with self-recorded or non-publicly available data.



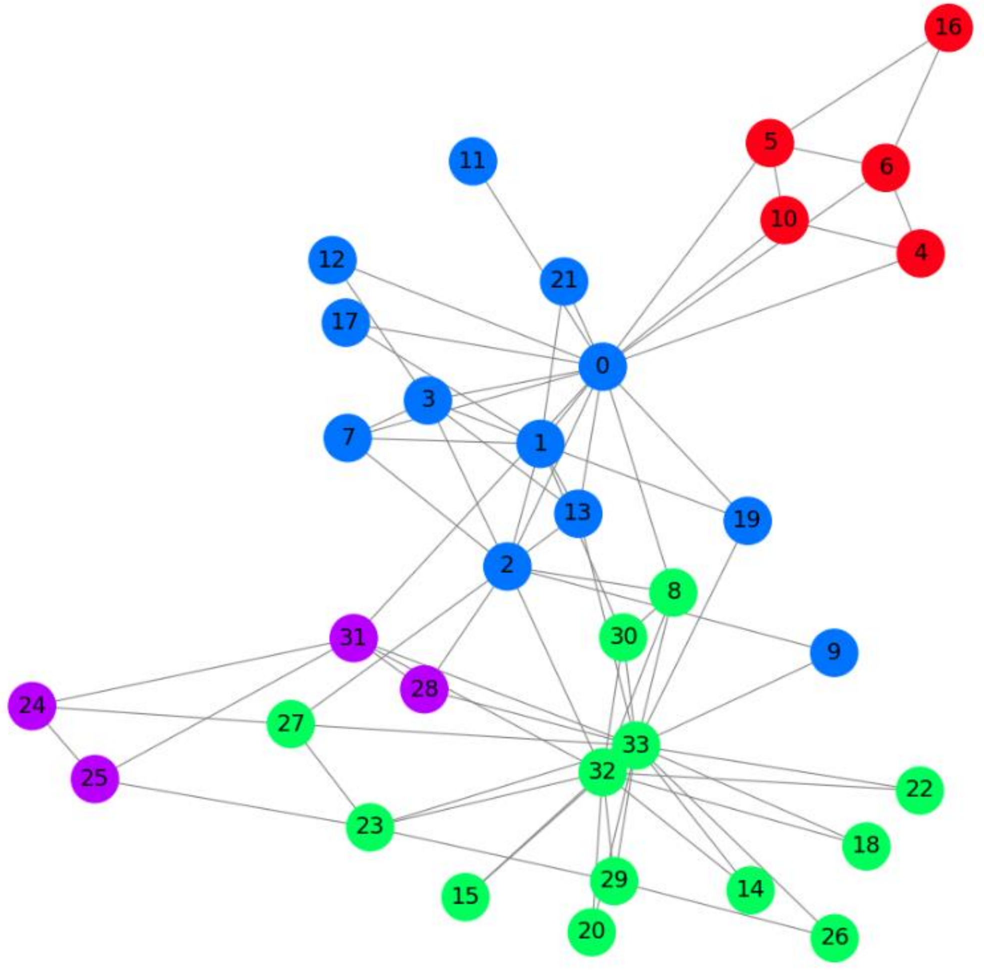
```
class MyDataset(InMemoryDataset):
    def __init__(self, root, data_list, transform=None):
        self.data_list = data_list
        super().__init__(root, transform)
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def processed_file_names(self):
        return 'data.pt'

    def process(self):
        torch.save(self.collate(self.data_list), self.processed_paths[0])
```

PyTorch Geometric: Creating our own Graph Datasets

Example: Creating a Karate Graph



```
[14]: class KarateClub(InMemoryDataset):
          def __init__(self, transform: Optional[Callable] = None):
              super().__init__('.', transform)

              row = [
                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
                  1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4,
                  5, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 8, 9, 9, 10, 10,
                  10, 11, 12, 12, 13, 13, 13, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17,
                  18, 18, 19, 19, 19, 20, 20, 21, 21, 22, 22, 23, 23, 23, 23, 23, 24,
                  24, 24, 25, 25, 25, 26, 26, 27, 27, 27, 27, 28, 28, 28, 29, 29, 29,
                  29, 30, 30, 30, 30, 31, 31, 31, 31, 31, 31, 32, 32, 32, 32, 32, 32,
                  32, 32, 32, 32, 32, 32, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33, 33,
                  33, 33, 33, 33, 33, 33
              ]
              col = [
                  1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 17, 19, 21, 31, 0, 2, 3, 7,
                  13, 17, 19, 21, 30, 0, 1, 3, 7, 8, 9, 13, 27, 28, 32, 0, 1, 2, 7,
                  12, 13, 0, 6, 10, 0, 6, 10, 16, 0, 4, 5, 16, 0, 1, 2, 3, 0, 2, 30,
                  32, 33, 2, 33, 0, 4, 5, 0, 0, 3, 0, 1, 2, 3, 33, 32, 33, 32, 33, 5,
                  6, 0, 1, 32, 33, 0, 1, 33, 32, 33, 0, 1, 32, 33, 25, 27, 29, 32,
                  33, 25, 27, 31, 23, 24, 31, 29, 33, 2, 23, 24, 33, 2, 31, 33, 23,
                  26, 32, 33, 1, 8, 32, 33, 0, 24, 25, 28, 32, 33, 2, 8, 14, 15, 18,
                  20, 22, 23, 29, 30, 31, 33, 8, 9, 13, 14, 15, 18, 19, 20, 22, 23,
                  26, 27, 28, 29, 30, 31, 32
              ]
              edge_index = torch.tensor([row, col])

              y = torch.tensor([  # Create communities.
                  1, 1, 1, 1, 3, 3, 3, 1, 0, 1, 3, 1, 1, 1, 0, 0, 3, 1, 0, 1, 0, 1,
                  0, 0, 2, 2, 0, 0, 2, 0, 0, 2, 0, 0
              ])

              x = torch.eye(y.size(0), dtype=torch.float)

              # Select a single training node for each community
              # (we just use the first one).
              train_mask = torch.zeros(y.size(0), dtype=torch.bool)
              for i in range(int(y.max()) + 1):
                  train_mask[(y == i).nonzero(as_tuple=False)[0]] = True

              data = Data(x=x, edge_index=edge_index, y=y, train_mask=train_mask)

              self.data, self.slices = self.collate([data])
```



네트워크 과학 연구실
NETWORK SCIENCE LAB
가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA